MANAGING AND SUPPORTING MULTITHREADED RESOURCES FOR NATIVE CODE IN HETEROGENEOUS A MANAGED RUNTIME ENVIRONMENT

Field

[0001] The embodiments of the invention relate to multithreaded programming and thread resource managements.

Background

[0002] Multithreading is a common programming techniques often used to maximize the efficiency of computer programs by providing a tool to permit concurrency or multitasking. Threads are ways for a computer program to be divided into multiple and distinct sequences of programming instructions where each sequence is treated as a single task. By assigning different tasks to multiple threads, programmers can design a program in a way that multiple tasks are executed concurrently.

[0003] One of the essential functions of a thread is to manage resources used by a task. A particular portion of resource such as a system memory, e.g. random access memory (RAM), may be shared between multiple threads or used only by a single thread. Generally, allocation of resources by a requesting thread is required. Managing the particular portion of resource allocated by the requesting thread is referred to as the per-thread context management. For example, allocating and releasing the resources specifically used by a thread.

[0004] Another programming technique often used to enhance software development is to maximize program reuse and to minimize program rewrite. Conventionally, a program designed to be executed on specific operating systems, e.g. Windows™ 98, IA-32, etc., must be rewritten or at least encapsulated if the program is to be executed on a different platform. Programming language such as Java™ has attempted to introduce and deliver a write once and deploy anywhere mechanism. For example, the Java™ virtual machine is a platform that creates a virtual environment and enables

programming code such as Java™ to be executed independent of the underlying hardware, e.g. Itanium™, SPARC™, PA-RISC, etc. Programming code that may be executed independent of the underlying hardware may be referred to as the platform-independent code.

[0005] A conventional method to permit the execution of platform-independent code on any hardware platform may require a translator. The translator translates the platform-independent code to a native code understood by the specific platform. For example, in a managed runtime environment that supports platform-independent managed code such as Java bye code, a Java™ virtual machine may be used to translate the Java byte code to platform specific code. In this case, the Java™ virtual machine is a binary translator that translates the Java byte code to platform specific code.

[0006] Managing multiple threads in a platform-independent code can be challenging. Traditionally, each thread is created and managed by underlying supporting libraries. For example, a thread written in C programming language is created and managed by the standard C library or the standard thread library. The underlying supporting library allocates resources such as stack frame saved register status, control words, and per-thread local resource.

[0007] Conventionally, a thread is created by the following procedure. First, a parent thread calls a routine, e.g. CreateThread(), to create a child thread. CreateThread() can be an application programming interface (API) defined in a thread library. When the thread library receives this function call, the thread library allocates resources, such as the ones described above, for the new child thread. Subsequent to the allocation of child thread resources, the thread library sends a request to an operating system and requests a creation of the new child thread. This may be necessary because while the thread library may manage per-thread context, the operating system may manage the multiple threads operations. For example, the operating system manages the intercommunication between multiple threads.

[0008] After the operating system has successfully created the new child thread, the thread library will complete the new thread initialization by associating the resources

of a parent thread with the newly created child thread. The parent thread is referred to as the portion of the programming code that initiates the child thread creation. While the child thread may freely use the allocated resources, the parent thread needs to maintain an access to the child thread. This may be done by providing or retaining an access to the child thread from the parent thread and associating the resource allocated for the child thread with the parent thread.

[0009] After the child thread has completed its task, the thread is usually terminated and the resources are returned to a resource pool so other threads may use the resource. An API call to the thread library, e.g. TerminateThread() could initiate the termination process. When TermianteThread() is called, the thread library would de-allocate the resources that were allocated during the thread initialization process. Subsequent to the de-allocation of thread resource, the thread library sends a notification to the operation system and notifies an exit or termination of the child thread.

[0010] When executing threads on a different platform, the platform may be equipped with different underlying supporting libraries and may not support the native thread code. As a result, a child native thread may not have been created and initialized properly. For example, a thread that has not been initialized due to lack of proper underlying supporting libraries may not be able to access its local variable because data structures and memory of this local variable have not yet been properly allocated.

[0011] Another challenge in program reuse may occur when porting existing program from a slower processor environment (e.g. a 32-bit program) to a faster processor environment (e.g. a 64-bit platform). In this situation, the 32-bit program is conventionally executed on an instruction set architecture (ISA) that supports 32-bit program, e.g. ISA-32. On the other hand, the faster processor environment may support only ISA-64 specific programming code. When attempting to execute the 32-bit program on a 64-bit platform, a mix ISA Execution (MIE) has occurred.

Contents of the Invention

[0012] A multithreaded program including a mixture of thread codes such as the platform-independent thread code (e.g. Java™ byte code) and native thread code (e.g. C), may be ported and transferred to a new platform that may not support the native thread code. Conventionally, when a multithreaded program including a mixture of code is ported to a new platform, a dynamic binary translator may be used to translate the entire program including the platform-independent code such that the ported program may be executed on the new platform. However, translating the entire program may cause the program to be inefficient. Instead, the problem is how to manage and support per-thread context wherein the native thread code may be transparently initiated and created in the new platform.

[0013] One embodiment of the invention includes a method to initialize a native thread code from a new platform and suspend the initialization of the native code at a position where it can later be associated with a new thread created in the new platform.

[0014] One embodiment of the invention includes a method to create a new thread from a new platform and associate the resource allocated for a native thread with the new thread created in the new platform.

[0015] An advantage of the embodied solutions is that a multithreaded program including a mixture of platform independent code and native thread code may be executed on a new platform without rewriting the program.

Brief Description of the Diagrams

[0016] Various embodiments are illustrated by way of example and not by way of limitation in the figures of the accompanying drawings in which like references indicate similar elements. It should be noted that references to "an," "one," or "various" embodiments in this disclosure are not necessarily to the same embodiment, and such references mean at least one.

[0017] **Figure 1** depicts a system overview illustrating the porting of a multithreaded program according to an embodiment of the invention.

[0018] **Figure 2** depicts a dynamic binary translator according to an embodiment of the invention.

[0019] **Figure 3** depicts the encapsulated components of the dynamic binary translator depicted in Figure 2 according to an embodiment of the invention.

[0020] **Figure 4** depicts a flowchart demonstrating the foreign thread creation according to an embodiment of the invention.

[0021] **Figure 5** depicts a flowchart demonstrating a foreign thread termination procedure in accordance to an embodiment of the invention.

[0022] **Figure 6** is an overview of the operation that illustrates the creation process of a foreign thread according to an embodiment of the invention.

Detailed Descriptions

[0023] A method and apparatus for managing and supporting threads in multiple instruction set architectures (ISA) is described below. A person of ordinary skill in the pertinent art, upon reading the present disclosure, will recognize that various novel aspects and features of the present invention can implemented independently or in any suitable combination, and further, that the disclosed embodiments are merely illustrative and not meant to be limiting.

[0024] Figure 1 depicts a system overview in accordance of an embodiment of the invention. A multithreaded program 100 includes a foreign thread code 101. The foreign thread code 101 may be programming code that is written in a native programming language originally designed for a corresponding foreign platform. For example, a program designed to be supported by the Intel Architecture (IA) 32-bit platform may be referred to as a foreign code when the IA-32 program is ported or transferred to be executed on a IA-64 (e.g. a IA 64-bit) platform.

[0025] The multithreaded program 100 that includes the foreign thread code 101 may be transferred and ported to be executed on a host platform 150. In the example depicted in Figure 1, the multithreaded program 100 may be encapsulated with platform-independent code 105 wherein the encapsulating code or the platform-independent code 105 is supported by the underlying host platform 150. In this example, the platform-independent code may include the foreign thread code 102 which may be identical to the foreign thread code 101. For the purpose of illustration, only a foreign thread code 101 is discussed. It may be appreciated that since the multithreaded program 100 may also be transferred and executed on the host platform 150, without encapsulation of platform-independent code, the following discussions apply to the multithreaded program 100 as well.

[0026] The foreign thread code 101 may be designed with an expectation that the foreign thread code 101 is supported by a foreign platform 110. A portion of the foreign code 101 may used other libraries 111 and thread libraries 112. Furthermore, foreign

platform 110 may also include an operating system 113 and a foreign instruction set architecture (ISA) 114 that is understood by a foreign processor 115. An ISA is a specification of a set of all binary codes, also known as the opcodes or machine language that are commands in native forms to be understood by a particular computer processing unit (CPU). For example, the Intel™ 64-bit processor such as the Itanium Processor Family (IPF) and Extended Memory Technology (EM64T) processors may be designed with a particular ISA. The ISA supporting the Intel™ 64-bit processor may be referred to as ISA-64 and the ISA supporting the Intel™ 32-bit processor may be referred to as ISA-32.

[0027] The other libraries 111 may be any library that are used by the foreign thread code 101. For example, if part of the foreign thread code 101 requires a sorting of a data structure, the other libraries 111 may include a sort library. If part of the foreign thread code 101 requires mathematically computation, the other libraries 111 may include a math library.

[0028] After the foreign thread code 101 is transferred to the host platform 150, the foreign thread code 102 may have to rely on the other libraries 151 to supported the specific library function calls as discussed above. In addition, the foreign thread code 102 may also rely on thread libraries 152 for proper thread initialization, creation, and termination. For example, the thread libraries 152 may be a foreign thread library that is capable of managing and supporting the foreign thread code 102.

[0029] Furthermore, the host platform 150 may also include an operating system 153, a host ISA 154, and a host processor 155. An example of the foreign processor 115 and the foreign ISA 114 may be the Intel™ 32-bit processor supporting ISA-32. An example of the host processor 155 and host ISA 154 may be the Intel™ IPF supporting ISA-64. An example of the thread libraries 112 may be the IA-32 library that supports the ISA-32 program or thread code (e.g. foreign thread code 101). An example of the thread libraries 152 may also be the IA-32 library that supports the foreign thread code 102.

[0030] In one embodiment of the invention, the foreign thread code 102 may be included in a platform-independent code 105 to ensure that the transferred code may be

properly executed on the operating system 153, the host ISA 154, and the host processor 155. For example, a Java™ application may contain foreign native code wherein the Java™ application is supported by the operating system 153 while the operating system 153 does not support the foreign native code contained within the Java™ application. It can be appreciated that a virtual machine, not shown in the figure, may be used in addition to the operating system 153 to ensure the operability of the Java™ application.

[0031] Figure 2 depicts a dynamic binary translator according to an embodiment of the invention. Platform such as the Java™ virtual machine supports platform-independent code. Java™ byte code is an example of the platform-independent code. However, Java™ byte code may not truly support transparent portability.

[0032] In an embodiment of the invention, a dynamic binary translator 203 may be used to ensure the program portability to a host platform 210. Similar to the configuration described in Figure 1, a platform-independent code 201 may encapsulate a foreign thread code 202. The foreign thread code 202 is translated by the dynamic binary translator 203 to a host native code 204. The host native code 204 may then be executed by the host platform 210 and supported by the underlying components such as other libraries 211, the thread libraries 212, an operating system 213, a host ISA 214, and a host processor 215. Here, the tread libraries 212 may be a foreign thread library capable of managing and supporting the foreign thread code 202.

[0033] Figure 3 depicts the encapsulated components of the dynamic binary translator depicted in Figure 2 according to an embodiment of the invention. In this illustrating, a multithreaded programming code 300 may represent both the original program before it is ported to a new platform and the program after it is transferred to be executed on a new platform. A dynamic binary translator 303 may include a middle tier layer 304. An application programming interface (API) 306 may be used as an interface between the multithreaded programming code 300 and the dynamic binary translator 303.

[0034] An embodiment of the invention may define a minimum set of APIs in the API 306 to be used for communication between the multithreaded programming code 300 and the dynamic binary translator 303. For example, mie_init_translator() and

mie_unload_translator() may be defined in the API 306 to initialize and release the dynamic binary translator 303. In addition, API calls such as mie_thread_init(), mie_complete_thread_init(), and mie_thread_term() may be defined in the API 306 to initialize new threads, completing the thread initialization and terminating the threads. Thread creation and termination function calls 301, such as mie_thread_init(), mie_complete_thread_init() and mie_thread_terms() may be called through the API 306 into the dynamic binary translator 303 and the middle tier layer 304.

[0035] In an embodiment of the invention, an operating system request 302 may be sent to the dynamic binary translator 303. Upon receiving the operating system request 302 by an operating system wrapper 305, the operating system request 302 may be suspended. The operating system request 302 may be initiated by the multithreaded programming code 300 to request a creation of a new thread. When the operating system wrapper 305 detects the request to create a new thread, instead of passing the operating system requests 302 down to an operating system 307, the operating system wrapper 305 may suspend the operating system request 302 and handle the thread creation function call later. The suspension may be needed because the new thread may require translation from a foreign code to a code supported by a host platform.

[0036] Figure 4 depicts a flowchart demonstrating the foreign thread creation process according to an embodiment of the invention. Parent thread 400 may call mie_thread_init() 401 to begin the process of creating a foreign thread in a host environment 490. During the initialization process, the mie_thread_init() 401 is received by a middle tier layer 410 and in response to the mie_thread_init() 401 function call. The middle tier layer 410 may initiate a function call to a foreign thread library 430 to allocate thread resources for the foreign thread in operation 402. In response to an allocation call from the middle tier layer 410, the foreign thread library 430 may begin the allocation of the thread resources. As discussed above, thread resources may include stack frame, register status, control words, and per-thread context storage.

[0037] Subsequent to the resources allocation performed by the foreign thread library 430, the parent thread 400 may initiate an operating system request 403 and

request a host operating system 450 to create a new thread. However, since the foreign thread may not have been properly initialized in a host environment 490, an operating system wrapper 440 may be used to intercept the operating system requests 403 and suspend the operating system request 403 until the proper initialization is performed.

[0038] In one embodiment of the invention, the parent thread 400 may call CreateThread() 404 to create a new thread in the host environment 490. This function call may be received by a host thread library 420. Similar to the process of initializing the foreign thread as discussed above, the host thread library 420 may begin resource allocations for the new thread. Subsequently, an operating system request may be requested by the parent thread 400 to the host operating system 450 to create the new child in operation 405. Since this thread is initialized by the host thread library 420, the initialization and the allocation of the resources is assumed to be properly performed. Therefore, a child thread 460 may be immediately created by the host operating system 450.

[0039] After the child thread 460 is properly created, the parent thread 400 may call mie_complete_thread_init (PARENT) in operation 406. This function call may be executed to associate the parent thread and the resource allocated for the foreign thread. A flag "PARENT" may be provided to indicate that the thread completion is to be performed in the context of the parent thread 400. In essence, this function call may be executed to complete the foreign thread resource initialization. In one embodiment of the invention, the operating system request 403 is then processed at this point to complete the foreign thread creation in the parent thread 400 context.

[0040] A similar process is performed for the child thread 460 to associate the resources allocated or the foreign thread and the child thread 460. As discussed above, the child thread 460 is created as the result of the CreateThread() 404. In an embodiment of the invention, the child thread 460 calls mie_complete_thread_init(CHILD) in operation 407. It is the same function call as the one with respect to associating the parent thread and the foreign thread resources. In this situation, a flag "CHILD" may be provided to indicate that the thread completion is to be performed in the context of the

child thread 460. The operating system wrapper 440 may provide the starting point wherein the completion process may begin. As a result, the thread completion process associates the resources allocated for the foreign thread and the child thread 460.

[0041] An application that may use these operations is a migration and reuse of a 32-bit application, such as the IA-32 library to a new 64-bit Itanium Processor Family (IPF) platform. In this situation, a 64-bit multithread program may need to call a native code encapsulated in the IA-32 library. Therefore, the IA-32 library is treated as the foreign thread code for the purpose of this discussion.

[0042] In this example, the 64-bit multithread program calls an IPF operating system create an IA-32 thread. Initially, the program calls mie_thread_init() to notify the middle tier layer 410. In mie_thread_init(), resource is allocated for the IA-32 thread. Subsequently, the program sends an operating system request 403 to create the IA-32 thread. This request is intercepted by the operating system wrapper 440 and the mie_thread_init() call is returned to the middle tier layer 410 at this point. This point of suspension may be referred to as the clone point.

[0043] After the function call mie_thread_init() is executed, the program calls another function call to create a thread in the host environment 490. In this example, the program calls CreateThread() directly to the IPF to create an IPF thread. In the host environment 490, the IPF thread may be properly initiated and created.

[0044] To associate the IA-32 thread and the IPF thread, the program calls mie_complete_thread_init(PARENT) 406. In this function, the parent IPF thread completes its thread initialization by creating the IA-32 thread beginning from the clone point. Furthermore, the child IPF thread also complete its thread initialization by calling mie_complete_tread_init(CHILD) 406. In this function, the IA-32 thread is translated and the resource context also begins from the clone point until the execution returns at the middle tier layer 410.

[0045] Figure 5 depicts a flowchart demonstrating a foreign thread termination procedure in accordance to an embodiment of the invention. Thread code 501 may

initiate a function call such as mie_thread_term() 502 to begin a thread termination process. When a middle tier layer 503 receives the mie_thread_term() 502 function call, it sends another request to a foreign thread library 505 a request to de-allocate the thread resources, as shown in operation 504. The foreign thread library 505 may perform proper de-allocation procedure and the foreign thread library 505 may send a notification of thread termination 506 to an operation system wrapper 507. Once the operating system wrapper receives the termination request, it may terminate the foreign thread or mark the foreign thread an exit point.

[0046] Subsequently, the thread code 501 may call a function to terminate the child thread 460 created in the host environment 490. For example, the thread code 501 may call TerminateThread() 553 and the function call may be received by a host thread library 552. In responds to this function call, the host thread library 552 may de-allocate the thread resources allocated for the child thread 460. In addition, the host thread library 552 may send a notification of thread termination 551 to a host operating system 550. In responds to the notification of thread termination 551, the host operating system 550 may mark the child thread 460 as to have exited the code.

[0047] Figure 6 is an overview of the operation that illustrates the creation process of a foreign thread according to an embodiment of the invention. Operation 601 creates a foreign child thread. To create the foreign child thread, operation 602 sends a thread initialization request to a foreign library. When the foreign library receives the thread initialization request, resources for the foreign thread are allocated in operation 603. After the resource allocation process, an operating system request to create the foreign child thread is sent and upon the detection of this request, the request is suspended in operation 604.

[0048] Operation 605 records a position, Position A, when the suspension occurs. This position may be used to associate the resource allocated for the foreign child thread with a thread created in a host environment. In operation 606, a host child thread is created. In one embodiment of the invention, creating the host child thread may involve at least a two step process. First, resources may be allocated for the host child thread in

the host environment. For example, in operation 607, a thread initialization request may be sent to a host thread library and request an allocation of the resource (operation 608). Second, an operating system request may be sent to a host operating system in requesting a creation of the host child thread (operation 609).

[0049] Subsequent a successful completion of host child thread creation, the operating system returns thread information to a calling code in operation 610. In operation 611, the foreign child thread initialization process is completed starting from Position A from operation 605. In operation 612, the host child thread initialization process is completed also starting from Position A from operation 605.

[0050] An example of a pseudo code according to an embodiment of the invention in which the method to construct an additional dependency graph may be implemented is provided below.

[0051]

```
//In parent thread's execution:
{
  mie_thread_init();
  CreateThread((*child_thread_entry_code)());
  mie_complete_thread_init(PARENT);
}

//Upon entering child thread's entry code
child_thread_entry_code()
{
 mie_complete_thread_init(CHILD);
 run_child_code();
 mie_thread_term();
}
```

[0052] One embodiment of the invention may be implemented on a machine-readable medium. A machine-readable medium may include any mechanism for storing or transmitting information in a form readable by a machine (*e.g.*, a computer), not limited to Compact Disc Read-Only Memory (CD-ROMs), Read-Only Memory (ROMs),

Random Access Memory (RAM), Erasable Programmable Read-Only Memory (EPROM), and a transmission over the Internet.

[0053] Although the invention has been described in detail hereinabove, it should be appreciated that many variations and/or modifications and/or alternative embodiments of the basic inventive concepts taught herein that may appear to those skilled in the pertinent art will still fall within the spirit and scope of the present invention as defined in the appended claims. For example, although the invention has been described to manage foreign thread resources between IA-32 threads and IPF platform, those skilled in the pertinent art may use the embodiments disclosed herein on different host and foreign ISA system, e.g. PA_RISC with IPF, IA-32 with PA-RISC, etc.